Architecture and Design Document

Hangout

Revision History

Date	Description	Author	Comments
10/14/2019	Initial Draft	Antonio Sanchez, Brandon Pitcher, Eric Curlett, Dean Vo, Ashley Williams	Initial draft of document
10/15/2019	v1.5	Antonio Sanchez	Added Scope, description, and trade-off analysis
10/16/2019			Trade-off analysis moved to the end of document
12/09/2019	v2.0	Antonio Sanchez	Added Flux Design Pattern to Front End Architecture Details
12/10/2019	v2.1	Antonio Sanchez	Added Database trade off analysis and Architectural Pattern trade off analysis.

Document Approval

Feature Name	Printed Name	Title	Date
Final Revision	Dean Vo	Product Owner	12/10/19

Table of Contents

Introduction

1.1 Purpose

This architecture and design document is intended display the overview of implementation of *Hangout* at every level.

1.2 Scope

This design document is meant to provide an overview of the structure of our webapp. This document also includes the tradeoff analysis between different frameworks to choose from. UML Diagrams and Sequence Diagrams are included to show how each component interacts with each other.

1.3 Intended Audience

Hangout is intended as a lightweight web app that users will be able to quickly and efficiently meet with others of similar interests. Hangout is designed to be used by all types of users who might use social media, and thus must be designed to be accessible and quick over detailed and complicated.

1.4 Overview

Hangout will use a Decision Tree learning model inorder to serve relevant ads as well as suggestions for Hangout™s that we believe the users will likely respond well to and be interested in.

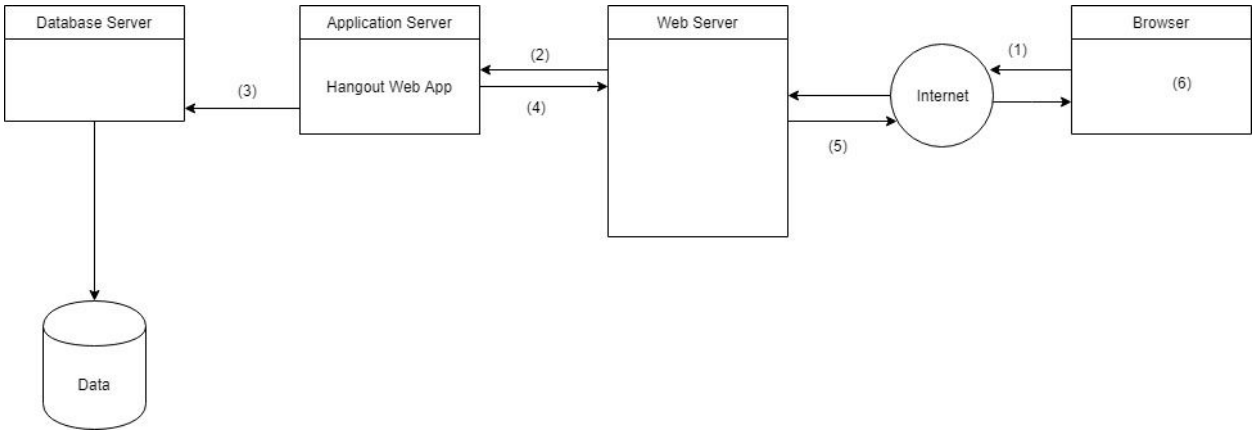
The server side code will be written in NodeJS, which is sufficiently lightweight and flexible for our needs as well as being relatively simple

We will have an Apache Web Server running a virtual console of Debian

React will be used as the front end. React is fast, scalable, and highly testable. It also is reportedly also very simple and easy to learn, which is appealing as we have limited time and capacity.

Architecture Details

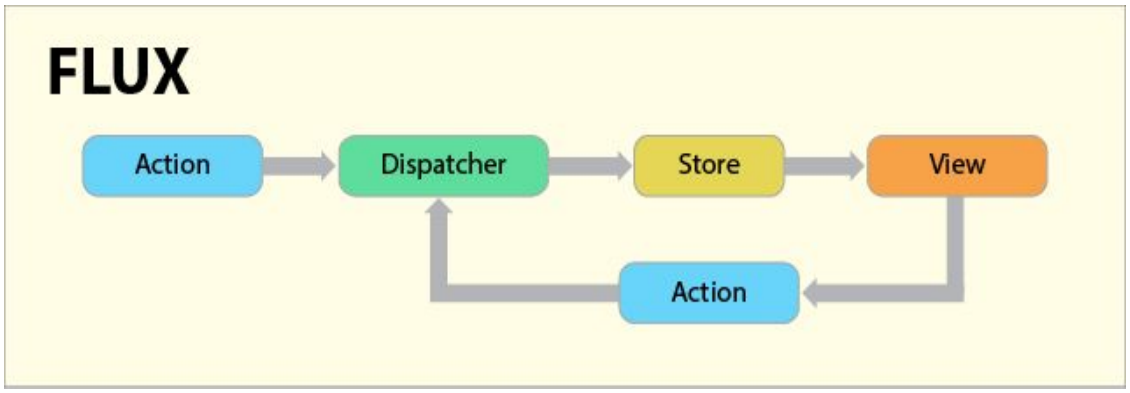
2.1 Top Level Architecture System Diagram



The diagram above is a block view of the *Hangout* system, displaying the different interactions between the modules. The numbered arrows show the flow of control between the modules. It behaves like a standard web app with the flow of control going from browser (1) to web server to our web app, and finally to our database server that stores our user info .

2.1.1 Front End Design Diagram (Flux)

In the diagrams below, we have decided to implement the Flux design pattern for front end components of the *Hangout* webapp.



The goal is to take advantage of the more stream-lined flow of data when dealing with multiple front end components. It provides a way to attach data to views as it gets more complex.
This is a pattern for managing how data flows through the React Application.

4 Main Components in Flux

Actions - Helpers that pass data to the Dispatcher

Dispatcher - Receives these Actions and broadcast payloads to registered callbacks.

Stores - Act as containers for application state & logic. The real work in the application is done in the Stores. The Stores registered to listen in on the actions of the Dispatcher will do accordingly and update the Views.

Controller Views - React Components grab the state from the stores and then pass it down to the child components.
The Controllers in the MVC and Flux are different. Here the Controllers are Controller-Views, and are found at the very top of the hierarchy. Views are React components.All the functionality is usually found within the Store. The Store is where all the work is done and tells the Dispatcher which events/actions it is listening for.

The Action, Dispatcher, Store and View are independent nodes with specific inputs and outputs. The data flows through the Dispatcher, the central hub, which in turn manages all the data.The Dispatcher acts as a registry with registered callbacks that the Stores respond to. Stores will emit a change which will be picked by the Controller-Views.

2.1.2 Code Example (Flux)

* All code will be pushed onto our github repository. For testing purposes, code implementing flux will be branched off into *Flux* branch until it's ready to be merged with the *master* branch.

Without Flux:

eventList.js

```
import React, { Component, PropTypes } from 'react';
import ListItem from './ListItem';
import ListHeader from './ListHeader';
import EmptyList from './EmptyList';

class List extends Component {
  getListOfItemIds = items => Object.keys(items)
 //Need an accumulator to get current list of
//events
  getTotalNumberOfListItems = items => (
    this.getListOfItemIds(items).reduce((accumulator, itemId) => (
      accumulator + parseInt(items[itemId].quantity, 10)
```

```
    ), 0)
  )

//list items here created but coupling is tight
  createListItemElements(items) {
    let item;
    const { removeListItem } = this.props;

    return (
      this
      .getListOfItemIds(items)
      .map(itemId => {
        item = items[itemId];
        return (<ListItem item={item} removeListItem={removeListItem} key={item.id}
/>);
      })
      .reverse()
    );
  }

  render() {
    const { items, removeAllListItems } = this.props;
    const listItemElements = this.createListItemElements(items);

    return (
      <div>
        <h3 className="page-header">

          <ListHeader
            totalNumberOfListItems={this.getTotalNumberOfListItems(items)}
            removeAllListItems={removeAllListItems}
          />

        </h3>
        <ul>

          {listItemElements.length > 0 ? listItemElements : <EmptyList />}

        </ul>
      </div>
    );
  }
}

List.propTypes = {
  removeListItem: PropTypes.func.isRequired,
  removeAllListItems: PropTypes.func.isRequired,
```

```
  items: PropTypes.array.isRequired,
};

export default List;
```

With Flux:

eventList.js

```
var React = require('react');
var ListItem = require('./ListItem');
var ListHeader = require('./ListHeader');
var EmptyList = require('./EmptyList');
var ListItemStore = require('../stores/ListItemStore');

var List = React.createClass({

  getInitialState: function () {
    return this.getList();
  },

  getList: function () {
    return {
      items: ListItemStore.getAllListItems()
    };
  },

  updateState: function () {
      this.setState(this.getList());
  },

  //components added to update state after Dispatcher calls
 //a store
  componentDidMount: function () {
    ListItemStore.addChangeListener(this.updateState);
  },

  componentWillUnmount: function () {
    ListItemStore.removeChangeListener(this.updateState);
  },

  getListOfItemIds: function (items) {
    return Object.keys(items);
  },

  getTotalNumberOfListItems: function (items) {
    var totalNumberOfItems = 0;
```

```
    var item;

    this.getListOfItemIds(items).forEach(function (itemId) {
      item = items[itemId];
      totalNumberOfItems = totalNumberOfItems + parseInt(item.quantity, 10);
    });

    return totalNumberOfItems;
  },

  createListItemElements: function (items) {
    var item;

    return (
      this
      .getListOfItemIds(items)
      .map(function createListItemElement(itemId) {
        item = items[itemId];
        return (<ListItem item={item}
handleRemoveListItem={this.props.removeListItem} key={item.id} />);
      }.bind(this))
      .reverse()
    );
  },

  render: function () {
    var items = this.state.items;
    var listItemElements = this.createListItemElements(items);

    return (
      <div>
        <h3 className="page-header">

          <ListHeader totalNumberOfListItems={this.getTotalNumberOfListItems(items)}
/>

        </h3>
        <ul>

          {listItemElements.length > 0 ? listItemElements : <EmptyList />}

        </ul>
      </div>
    );
  }
});

module.exports = List;
```

Listitemstore.js

```
var Dispatcher = require('../dispatcher/Dispatcher');
var EventEmitter = require('events').EventEmitter;
var objectAssign = require('object-assign');

var eventList = {};
```

//all stores need for list items are now stored here rather than in separate files

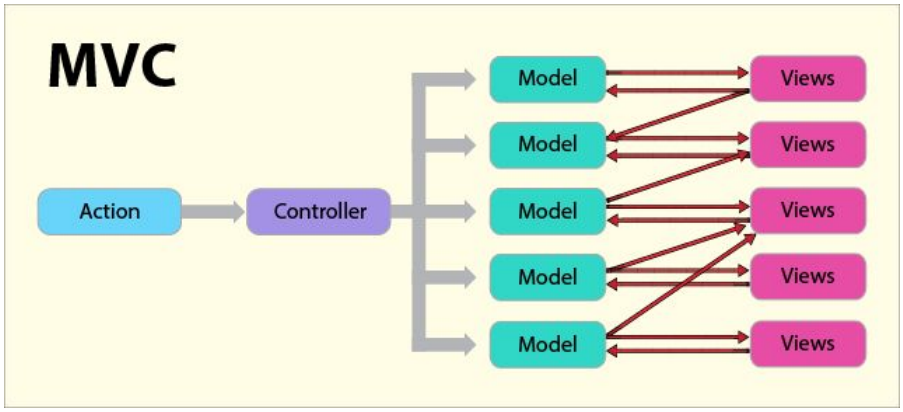
```
function addListItem(listItem) {
  eventList[listItem.id] = listItem;

  ListItemStore.emit('change');
}

function removeListItem(listItemId) {
  delete eventList[listItemId];

  ListItemStore.emit('change');
}

function removeAllListItems() {
  eventList = {};

  ListItemStore.emit('change');
}

var ListItemStore = objectAssign({}, EventEmitter.prototype, {

  getAllListItems: function () {
    return eventList;
  },

  addChangeListener: function (changeEventHandler) {
    this.on('change', changeEventHandler);
  },

  removeChangeListener: function (changeEventHandler) {
    this.removeListener('change', changeEventHandler);
  }

});

function handleAction(action) {
  if (action.type === 'add_list_item') {
```

```
    addListItem(action.item);
  } else if (action.type === 'remove_list_item') {
    removeListItem(action.itemId);
  } else if (action.type === 'remove_all_list_items') {
    removeAllListItems();
  }
}
```

```
//handleAction in this file calls for the dispatcher
ListItemStore.dispatchToken = Dispatcher.register(handleAction);
```

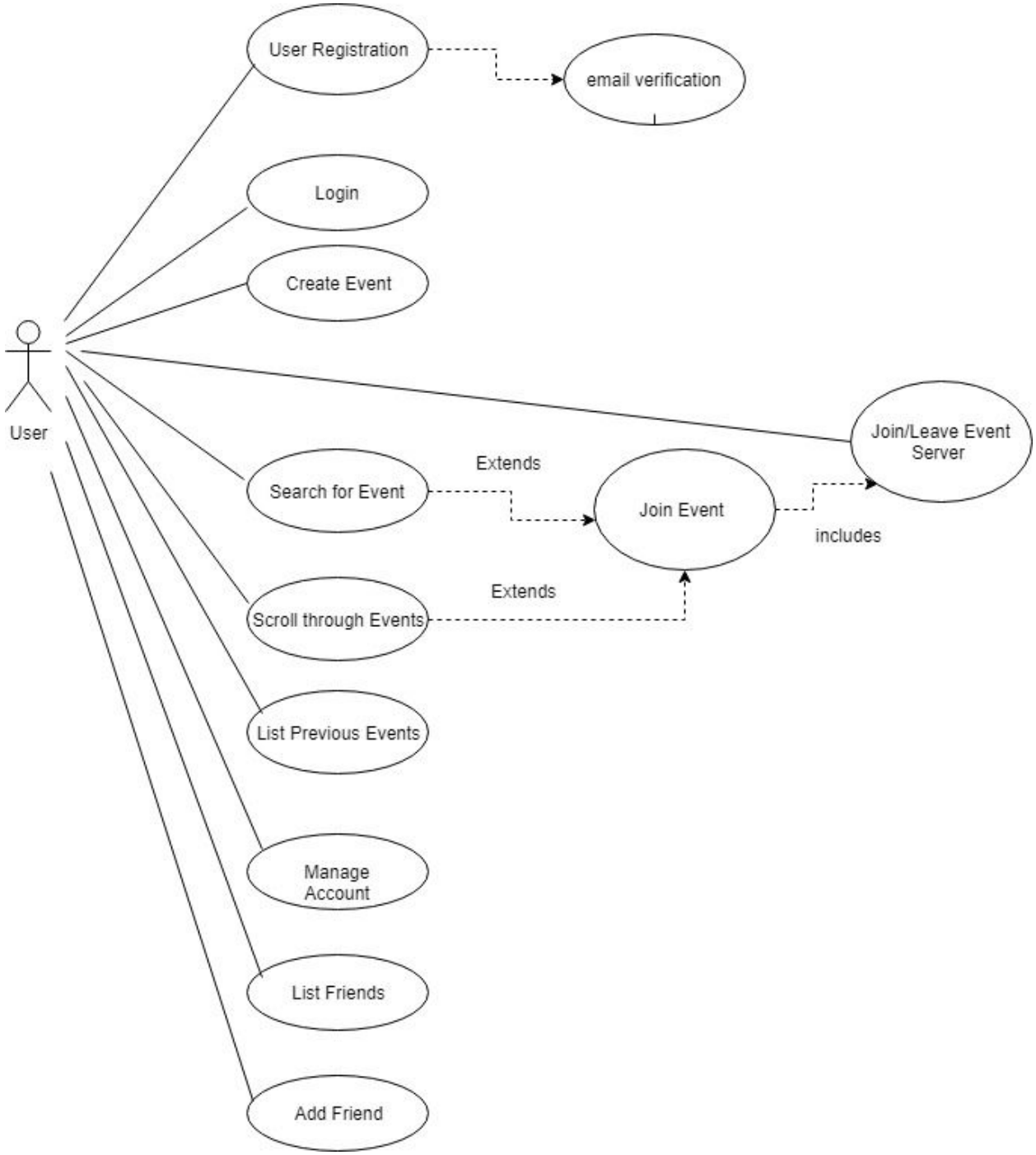
```
//dispatcher stored in separate file
module.exports = ListItemStore;
var Dispatcher = require('flux').Dispatcher;

module.exports = new Dispatcher();
```

2.1.2 Back End Design Diagram (MVC)



We will be using the traditional model-view-controller design pattern for the backend architecture design. Since the flow of data is considered more rigid when moving through server and client, MVC provides an efficient way to implement our back-end components.

2.2 Use Cases


User Registration
email verification
Login
Create Event
User
Join/Leave Event Server
Search for Event
Extends
Join Event
includes
Scroll through Events
Extends
List Previous Events
Manage Account
List Friends
Add Friend

2.2.1 Use Case UC1

2.2.1.1 *Objective*

Registration

2.2.1.2 *Priority*

High

2.2.1.3 *Actors*

End-User

2.2.1.4 *Pre-conditions*

The user is connected to the internet and navigates to the *Hangout* website on their browser.

2.2.1.5 *Post-conditions*

The user is now a member of *Hangout.*

2.2.2 Use Case UC2

2.2.2.1 *Objective*

Verification

2.2.2.2 *Priority*

Medium

2.2.2.3 *Actors*

End-User

2.2.2.4 *Pre-conditions*

The user is registering to *Hangout* as a new user.

2.2.2.5 *Post-conditions*

User is now a registered user of *Hangout* and create/search for Events and update their account.

2.2.3 Use Case UC3

2.2.3.1 *Objective*

Login

2.2.3.2 *Priority*

High

2.2.3.3 *Actors*

End-User

2.2.3.4 *Pre-conditions*

The user is connected to the internet and navigates to the *Hangout* website on their browser.

2.2.3.5 *Post-conditions*

User signs in and has access to the *Hangout* home page.

2.2.4 Use Case UC4

2.2.4.1 *Objective*

Create Event

2.2.4.2 *Priority*

High

2.2.4.3 *Actors*

End-User

2.2.4.4 *Pre-conditions*

A user is logged into *Hangout*.

2.2.4.5 *Post-conditions*

User has posted a new Event to the site.

2.2.5 Use Case UC5

2.2.5.1 *Objective*

Search Event

2.2.5.2 *Priority*

High

2.2.5.3 *Actors*

End-User

2.2.5.4 *Pre-conditions*

User logged into *Hangout*

2.2.5.5 *Post-conditions*

User is given a list of Events that match their search criteria.

2.2.6 Use Case UC6

2.2.6.1 *Objective*

Join Event

2.2.6.2 *Priority*

High

2.2.6.3 *Actors*

End-User

2.2.6.4 *Pre-conditions*

User is given an event list that matches his search criteria.

2.2.6.5 *Post-conditions*

User Joins an event.

2.2.7 Use Case UC7

2.2.7.1 *Objective*

Scroll through events

2.2.7.2 *Priority*

Medium

2.2.7.3 *Actors*

End-User

2.2.7.4 *Pre-conditions*

User can navigate through events held on a given day.

2.2.7.5 *Post-conditions*

The user is now a member of *Hangout.*

2.2.8 Use Case UC8

2.2.8.1 *Objective*

Join/Leave Event Server

2.2.8.2 *Priority*

Medium

2.2.8.3 *Actors*

End-User

2.2.8.4 *Pre-conditions*

User joins event.

2.2.8.5 *Post-conditions*

User is added to a chat with other users that joined the same event. User can also leave or mute the chat.

2.2.9 Use Case UC9

2.2.9.1 *Objective*

List Previous Events

2.2.9.2 *Priority*

Low

2.2.9.3 *Actors*

End-User

2.2.9.4 *Pre-conditions*

User is at their home page.

2.2.9.5 *Post-conditions*

User can see previous events they participated in.

2.2.10 Use Case UC10

2.2.10.1 *Objective*

Manage Account

2.2.10.2 *Priority*

High

2.2.10.3 *Actors*

End-User

2.2.10.4 *Pre-conditions*

User is logged in and in their home page.

2.2.10.5 *Post-conditions*

User account has been updated as per their requirements.

2.2.11 Use Case UC11

2.2.11.1 *Objective*

List Friends

2.2.11.2 *Priority*

Low

2.2.11.3 *Actors*

End-User

2.2.11.4 *Pre-conditions*

2.2.11.5 *Post-conditions*

2.2.12 Use Case UC12

2.2.12.1 *Objective*

Add Friends

2.2.12.2 *Priority*

Low

2.2.12.3 *Actors*

End-User

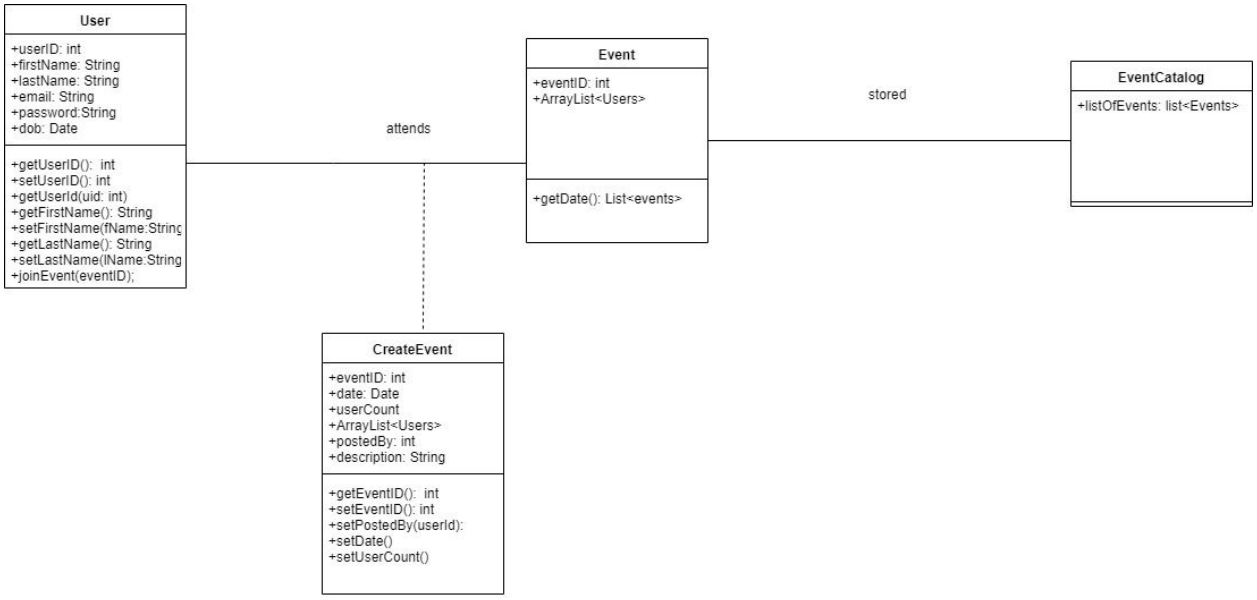
2.2.12.4 *Pre-conditions*

User has finished an event.

2.2.12.5 *Post-conditions*

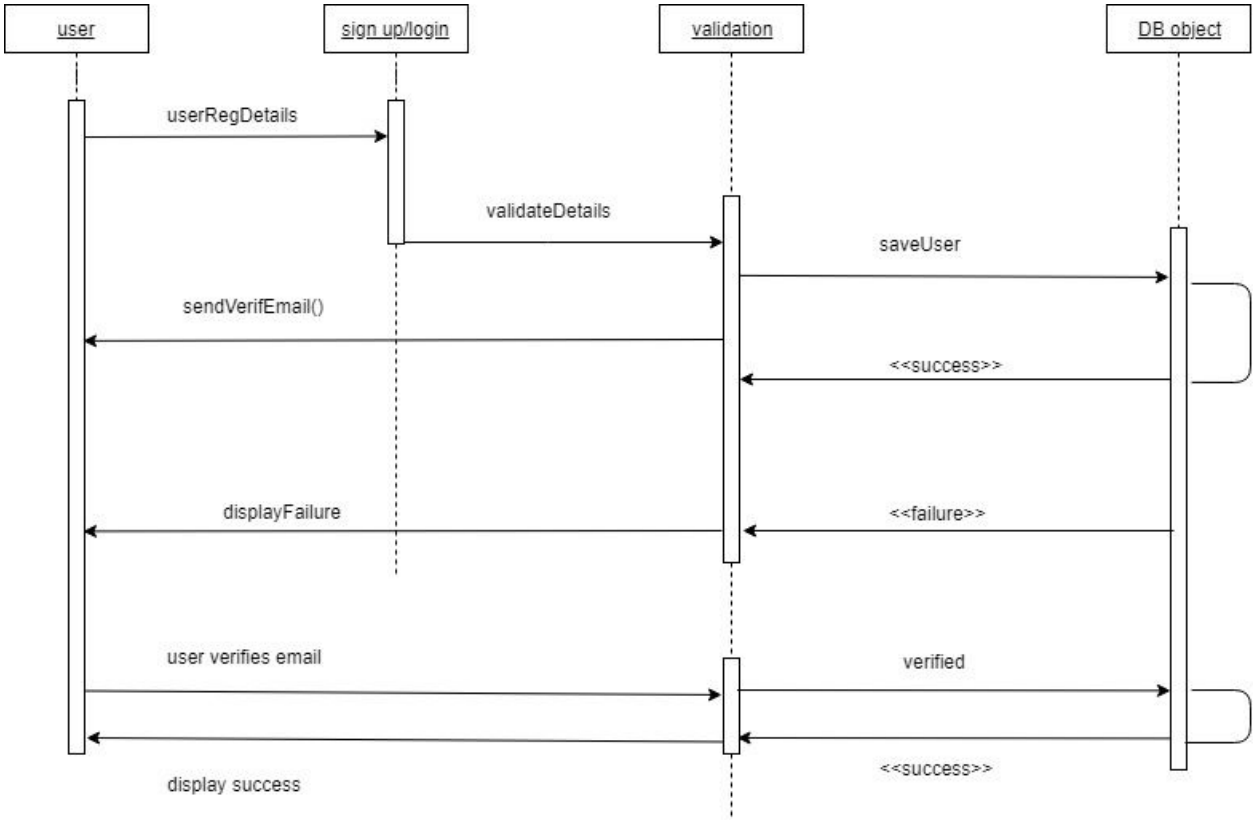
User can add other Users from past events to friends list.

2.3 Class Diagram

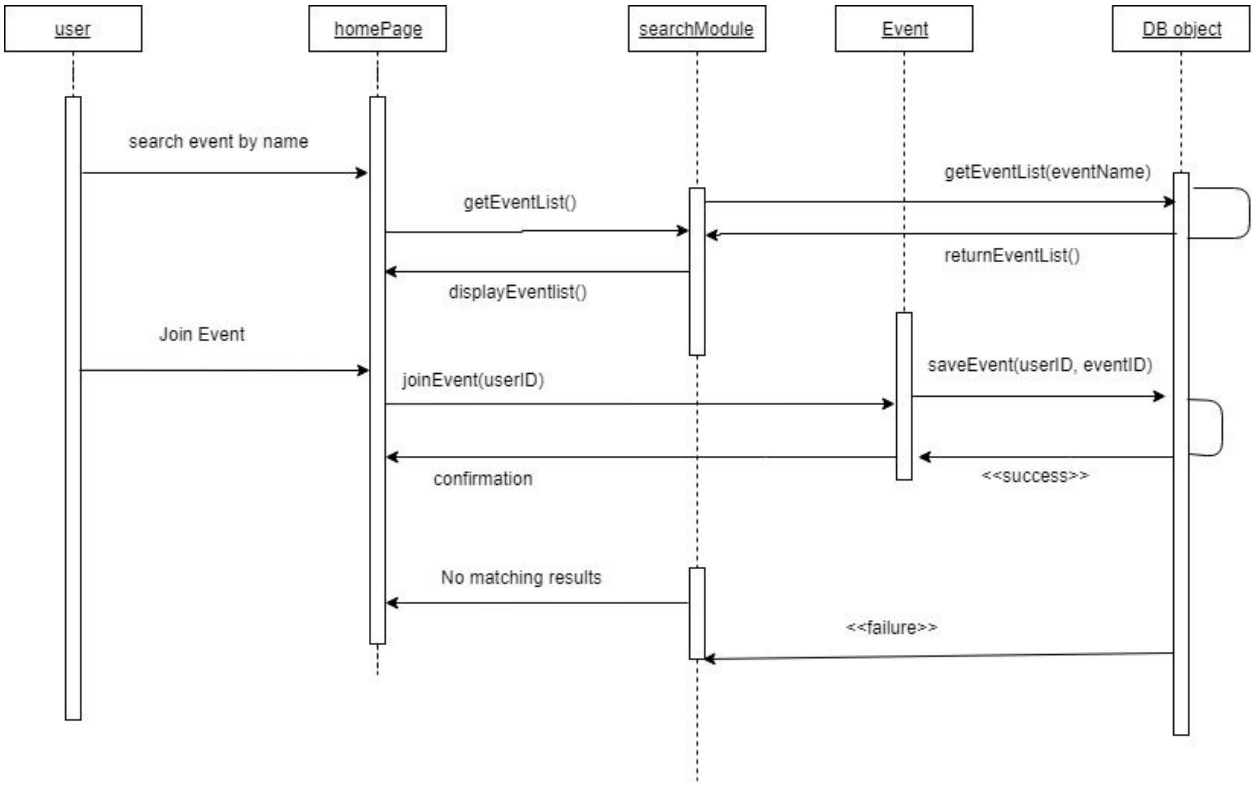


2.4 Sequence Diagrams

2.4.1 Registration



2.4.2 Search for Event



Analysis

3.1 Tradeoff analysis

Front end	Pros	Cons
React	Familiarity, easy to learn, Fast, supports server-side rendering, Functional programming	Less standardized, Less class based, Mixes templating with JSX logic
Angular	Supports Typescript, prebuilt npm libraries, One way data binding	Variety of structures, slower than others
Vue	Adaptable, offers strong integration tools, allows for large templates, low weight	Small market share means finding resources is harder. Less global experience as well as team experience.

Backend	Pros	Cons
Node	Non-blocking I/O, single language, Flexible, familiarity, JSON focused, speed	Single threaded, Complexity
PHP	Massively popular and well supported, supports relational databases	Slower, combines html with it resulting in confusion, not component modular
Java	Super familiar, powerful, cross platform	Most server plugins require payment
Linode (virtual server)	affordable, transparent, scalable, reliable, full access to server, great customer service w/ phone line support, predictable pricing, open cloud (no vendor lock in), IP address of virtual server does not change	only supports Linux operating systems
Express	most popular web framework on NPM, provides simple user authentication, allows for login using social media, opensource	single threaded framework, poor scalablity

* Options that were underlined were our choices to proceed development with for the *Hangout* webapp

Database Trade-off Analysis

Database	Pros	Cons
MySQL	● We have prior experience with this DB ● It's available for free. ● It offers a lot of functionality even for a free database engine. ● There are a variety of user interfaces that can be implemented. ● It can be made to work with other databases, including DB2 and Oracle.	● Time-consuming to set up ● Support not available in free version.
MongoDB	● It's fast and easy to use. ● The engine supports JSON and other NoSQL documents. ● Data of any structure can be stored and accessed quickly and easily. ● Schema can be written without downtime.	● SQL is not used as a query language. ● Tools to translate SQL to MongoDB queries are available, but they add an extra step to using the engine. ● Setup can be a lengthy process. ● Default settings are not secure.
Microsoft SQL Server	● It is very fast and stable. ● The engine offers the ability to adjust and track performance levels, which can reduce resource use. ● You are able to access visualizations on mobile devices. ● Works very well with other Microsoft products.	● Resource wasting ● Many individuals have issues using the SQL Server Integration Services to import files.
PostGRE SQL	● This database management engine is scalable and can	

	handle terabytes of data. ● It supports JSON. ● There are a variety of predefined functions. ● A number of interfaces are available.	

Database Chosen: MongoDB

We chose to work with this database for now because we didn't find the lack of SQL to be a big deal. We will use this to take advantage of the Mongoose Express middleware.

Architectural Pattern Tradeoff Analysis

Listed below are some of the architectural patterns we look to employ for *Hangout.* As the app becomes more complex, different problems might require a different pattern. Where we would plan to implement the pattern is written inside the parenthesis.

Architecture Pattern	Pros	Cons
Layered (Front-End)	● Standardization is easier ● Changes can be made within layer ● Lower layer can be used by different higher layers	● Not universally applicable ● May have to skip layers in certain situations
Flux (Front-End Implementation)	● Unidirectional Data flow ● Centered on react ● 4 components	● Restricted to front end
Client - Server (Full Stack)	● Standard web app model ● reliable	● Requests typically handled in separate threads on the server ● Intra-process communication causes overhead as different clients have different representations

Model View Controller (Full Stack Implementation)	• Multiple views possible for same model • Can be disconnected or connected at run time	• Increases complexity • Can lead to unnecessary updates
Event-Bus (full stack)	• Easy connections to publishers, subscribers, and connections • Effective for highly distributed applications	• Scalability is a problem • All messages travel through the same event bus
Peer to Peer (specifically for messaging)	• Supports decentralized computing • Robust • Scalable	• No guarantee of quality of service • Can't guarantee security • Performance depends on the number of nodes
Master-slave (Full Stack)	• Efficient delegation • Accuracy	• Isolated slaves; no shared state • Latency too high

Architecture to be implemented: Flux (front-end)/MVC (back-end)

We chose to work with the Flux Architecture pattern for the front-end because it's currently being used by Facebook, a site that we are trying to model *Hangout* after. It's an architecture/pattern that allows us to control the flow of data.

We will proceed to model our back-end in the Model View Controller architecture. Since data flow is already rigid in the back end, MVC is the standard choice for us to model our data.